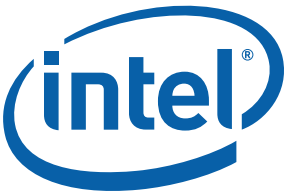

Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

March 2007

Notice: The Intel® 64 and IA-32 architectures may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-019

Contents

Revision History

Version	Description	Date
-001	• Initial Release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion.	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006
-017	• Removed Documentation Changes 1-23. • Added Documentation Changes 1-36.	September 2006
-018	• Added Documentation Changes 37-42.	October 2006
-019	• Removed Documentation Changes 1-42. • Added Documentation Changes 1-20.	March 2007

Preface

This document is an update to the specifications contained in the Affected Documents/ Related Documents table below. This document is a compilation of documentation changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents/Related Documents

Document Title	Document Number
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture	253665
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M	253666
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z	253667
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide.	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide	253669

Nomenclature

Documentation Changes include errors or omissions from the current published specifications. These changes will be incorporated in the next release of the Software Developer's Manual.

Summary Table of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. This table uses the following notations:

Codes Used in Summary Table

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

Summary Table of Documentation Changes

Number	Documentation Changes
1	APIC ID reference corrected
2	VMPTRST summary table correction
3	MOV to/from Debug Registers, opcode correction
4	Blocks of pseudocode updated
5	Material covering handling of VM Exit during Virtual-NMI injection corrected
6	More information about R8-15 & XMM8-15 transitions
7	Figure 8-6 corrected
8	Note added to section on APIC timer
9	MOV CR opcode table corrected
10	Coverage of PEBS updated
11	Missing exception added for MFENCE
12	IA32_MCG_STATUS information added
13	Introduction section for CPUID updated
14	Update to CPUID documentation on deterministic cache parameters leaf
15	Updated pseudocode in VMCALL description
16	IA32_MCi_STATUS figure corrected
17	IA32_MCi_STATUS flag description updated
18	Instruction summaries fixed for MOVD/MOVQ, PMOVMSKB, PINSRW, PEXTRW
19	Correction to microcode update documentation
20	PSHUFB compiler intrinsic fixed

Documentation Changes

1. APIC ID reference corrected

In Section 7.7.5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, an APIC ID reference has been corrected.

7.7.5 Identifying Logical Processors in an MP System

After the BIOS has completed the MP initialization protocol, each logical processor can be uniquely identified by its local APIC ID. Software can access these APIC IDs in either of the following ways:

- **Read APIC ID for a local APIC** — Code running on a logical processor can execute a MOV instruction to read the processor's local APIC ID register (see Section 8.4.6, "Local APIC ID"). This is the ID to use for directing physical destination mode interrupts to the processor.
- **Read ACPI or MP table** — As part of the MP initialization protocol, the BIOS creates an ACPI table and an MP table. These tables are defined in the Multiprocessor Specification Version 1.4 and provide software with a list of the processors in the system and their local APIC IDs. The format of the ACPI table is derived from the ACPI specification, which is an industry standard power management and platform configuration specification for MP systems.
- **Read Initial APIC ID** — An APIC ID is assigned to a logical processor during power up and is called the initial APIC ID. This is the APIC ID reported by CPUID.1:EBX[31:24] and may be different from the current value read from the local APIC. Use the initial APIC ID to determine the topological relationship between logical processors.

  Bits in the initial APIC ID can be interpreted using several bit masks. Each bit mask can be used to extract an identifier to represent a hierarchical level of the multi-threading resource topology in an MP system (See Section 7.10.1, "Hierarchical Mapping of Shared Resources"). The initial APIC ID may consist of up to four bit-fields. In a non-clustered MP system, the field consists of up to three bit fields.

... ... more text here... ...

2. VMPTRST summary table correction

In Section "VMPTRST—Store Pointer to Virtual-Machine Control Structure" in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, the summary table has been corrected. See the corrected cells below.

VMPTRST—Store Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /7	VMPTRST m64	Stores the current VMCS pointer into memory.

... ... more text here ... ...

3. MOV to/from Debug Registers, opcode correction

In Section "MOV—Move to/from Debug Registers" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, the summary table has been corrected. See the corrected table cells below.

MOV—Move to/from Debug Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 21/*r*	MOV *r32*, DR0-DR7	N.E.	Valid	Move debug register to *r32*
0F 21/*r*	MOV *r64*, DR0-DR7	Valid	N.E.	Move extended debug register to *r64*.
0F 23 /*r*	MOV DR0-DR7, *r32*	N.E.	Valid	Move *r32* to debug register
0F 23 /*r*	MOV DR0-DR7, *r64*	Valid	N.E.	Move *r64* to extended debug register.

... ... more text here ... ...

4. Blocks of pseudocode updated

In Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, some of the pseudocode has been updated. There are multiple sections, identified by the reproduced code blocks below.

Three blocks of pseudocode were corrected in the "PCMPEQB/PCMPEQW/PCMPEQD—Compare Packed Data for Equal" section. Corrected blocks follow.

... ...

```
PCMPEQB instruction with 128-bit operands:
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
    IF DEST[127:120] = SRC[127:120]
        THEN DEST[127:120] ← FFH;
        ELSE DEST[127:120] ← 0; FI;
```

... ...

```
PCMPEQW instruction with 128-bit operands:
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th words in DEST and SRC *)
    IF DEST[127:112] = SRC[127:112]
        THEN DEST[127:112] ← FFFFH;
        ELSE DEST[127:112] ← 0; FI;
```

.. ..

```
PCMPEQD instruction with 128-bit operands:
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
    IF DEST[127:96] = SRC[127:96]
        THEN DEST[127:96] ← FFFFFFFFH;
        ELSE DEST[127:96] ← 0; FI;

… … …
```

Three blocks of pseudocode were corrected in the "PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than" section. Corrected blocks follow.

.. … …

```
PCMPGTB instruction with 128-bit operands:
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
    IF DEST[127:120] > SRC[127:120]
        THEN DEST[127:120] ← FFH;
        ELSE DEST[127:120] ← 0; FI;

… … …

PCMPGTW instruction with 128-bit operands:
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th words in DEST and SRC *)
    IF DEST[63:48] > SRC[127:112]
        THEN DEST[127:112] ← FFFFH;
        ELSE DEST[127:112] ← 0; FI;

… … …

PCMPGTD instruction with 128-bit operands:
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
    IF DEST[127:96] > SRC[127:96]
        THEN DEST[127:96] ← FFFFFFFFH;
        ELSE DEST[127:96] ← 0; FI;

… … …
```

5. Material covering handling of VM Exit during Virtual-NMI injection corrected

In Section 25.7.1.2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, some VM-exit material has been corrected. See the reproduced section below noted by change bars.

25.7.1.2 Resuming Guest Software after Handling an Exception

If the VMM determines that a VM exit was caused by an exception due to a condition established by the VMM itself, it may choose to resume guest software after removing the condition. The approach for removing the condition may be specific to the VMM's software architecture. and algorithms This section describes how guest software may be resumed after removing the condition.

In general, the VMM can resume guest software simply by executing VMRESUME. The following Items provide details of cases that may require special handling:

- If the "NMI exiting" VM-execution control is 0, bit 12 of the VM-exit interruption-information field indicates that the VM exit was due to a fault encountered during an execution of the IRET instruction that unblocked non-maskable interrupts (NMIs). In particular, it provides this indication if the following are both true:
  - — Bit 31 (valid) in the IDT-vectoring information field is 0.
  - — The value of bits 7:0 (vector) of the VM-exit interruption-information field is not 8 (the VM exit is not due to a double-fault exception).

  If both are true and bit 12 of the VM-exit interruption-information field is 1, NMIs were blocked before guest software executed the IRET instruction that caused the fault that caused the VM exit. The VMM should set bit 3 (blocking by NMI) in the interruptibility-state field (using VMREAD and VMWRITE) before resuming guest software.
- If the "virtual NMIs" VM-execution control is 1, bit 12 of the VM-exit interruption-information field indicates that the VM exit was due to a fault encountered during an execution of the IRET instruction that removed virtual-NMI blocking. In particular, it provides this indication if the following are both true:
  - — Bit 31 (valid) in the IDT-vectoring information field is 0.
  - — The value of bits 7:0 (vector) of the VM-exit interruption-information field is not 8 (the VM exit is not due to a double-fault exception).

  If both are true and bit 12 of the VM-exit interruption-information field is 1, there was virtual-NMI blocking before guest software executed the IRET instruction that caused the fault that caused the VM exit. The VMM should set bit 3 (blocking by NMI) in the interruptibility-state field (using VMREAD and VMWRITE) before resuming guest software.
- Bit 31 (valid) of the IDT-vectoring information field indicates, if set, that the exception causing the VM exit occurred while another event was being delivered to guest software. The VMM should ensure that the other event is delivered when guest software is resumed. It can do so using the VM-entry event injection described in Section 22.5 and detailed in the following paragraphs:
  - — The VMM can copy (using VMREAD and VMWRITE) the contents of the IDT-vectoring information field (which is presumed valid) to the VM-entry interruption-information field (which, if valid, will cause the exception to be delivered as part of the next VM entry).

- The VMM should ensure that reserved bits 30:12 in the VM-entry interruption-information field are 0. In particular, the value of bit 12 in the IDT-vectoring information field is undefined after all VM exits. If this bit is copied as 1 into the VM-entry interruption-information field, the next VM entry will fail because the bit should be 0.
- If the “virtual NMIs” VM-execution control is 1 and the value of bits 10:8 (interruption type) in the IDT-vectoring information field is 2 (indicating NMI), the VM exit occurred during delivery of an NMI that had been injected as part of the previous VM entry. In this case, bit 3 (blocking by NMI) will be 1 in the interruptibility-state field in the VMCS. The VMM should clear this bit; otherwise, the next VM entry will fail (see Section 22.3.1.5).

— The VMM can also copy the contents of the IDT-vectoring error-code field to the VM-entry exception error-code field. This need not be done if bit 11 (error code valid) is clear in the IDT-vectoring information field.

— The VMM can also copy the contents of the VM-exit instruction-length field to the VM-entry instruction-length field. This need be done only if bits 10:8 (interruption type) in the IDT-vectoring information field indicate either software interrupt, privileged software exception, or software exception.

6. More information about R8-15 & XMM8-15 transitions

In Section 3.4.1.1 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, information covering mode transition behavior for R8-15 and XMM8-15 has been added. This information has been reproduced in context below noted by change bars.

3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

Table 3-2. Addressable General Purpose Registers

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W

Table 3-2. Addressable General Purpose Registers

Register Type	Without REX	With REX
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

In 64-bit mode, there are limitations on accessing byte registers. An instruction cannot reference legacy high-bytes (for example: AH, BH, CH, DH) and one of the new byte registers at the same time (for example: the low byte of the RAX register). However, instructions may reference legacy low-bytes (for example: AL, BL, CL or DL) and new byte registers at the same time (for example: the low byte of the R8 register, or RBP). The architecture enforces this limitation by changing high-byte references (AH, BH, CH, DH) to low byte references (BPL, SPL, DIL, SIL: the low 8 bits for RBP, RSP, RDI and RSI) for instructions using a REX prefix.

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not be modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Because the upper 32 bits of 64-bit general-purpose registers are undefined in 32-bit modes, the upper 32 bits of any general-purpose register are not preserved when switching from 64-bit mode to a 32-bit mode (to protected mode or compatibility mode). Software must not depend on these bits to maintain a value after a 64-bit to 32-bit mode switch.

7. Figure 8-6 corrected

In Figure 8-6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, bit designations have been corrected. See the corrected figure below.

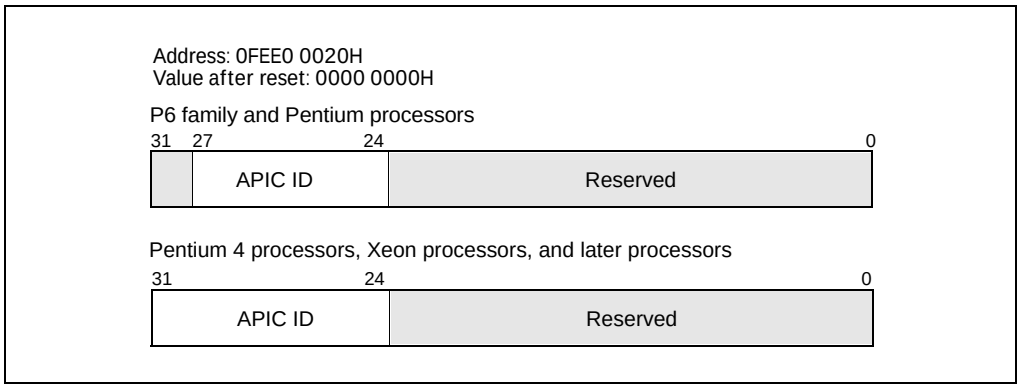

Figure 8-6. Local APIC ID Register

8. Note added to section on APIC timer

In Section 8.5.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a note has been added (discusses deep C-states and GV3 transitions). Part of the section is reproduced below with the change in context noted by change bar.

8.5.4 APIC Timer

The local APIC unit contains a 32-bit programmable timer that is available to software to time events or operations. This timer is set up by programming four registers: the divide configuration register (see Figure 8-10), the initial-count and current-count registers (see Figure 8-11), and the LVT timer register (see Figure 8-8).

NOTE

The APIC timer stops incrementing when the processor is in deep C-states (C3 or higher). It also does not increment during GV3 transitions.

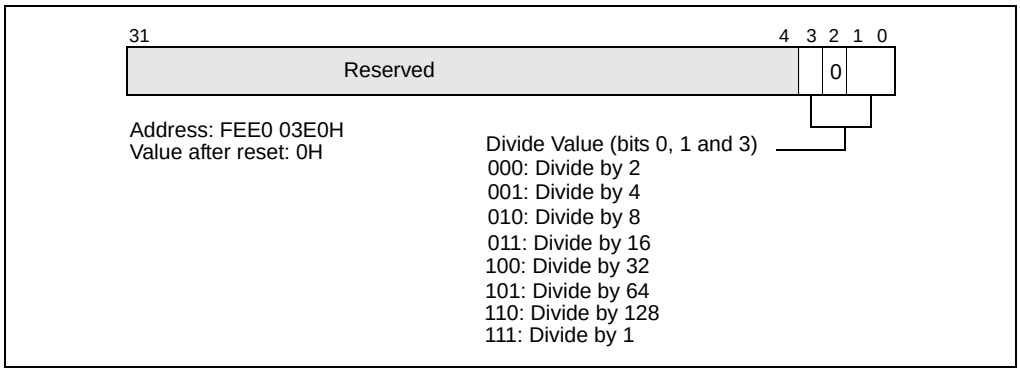

Figure 8-10. Divide Configuration Register

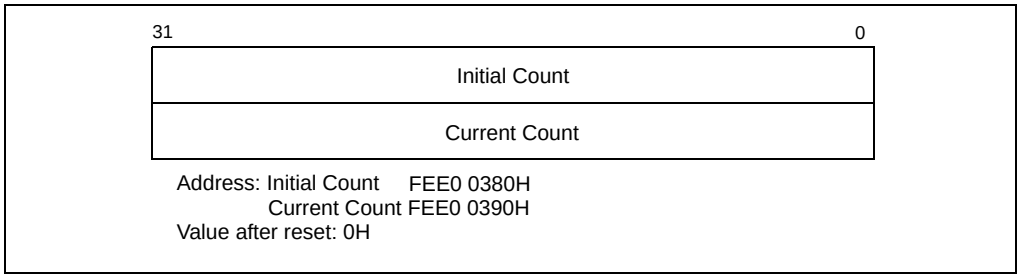

Figure 3-6. Initial Count and Current Count Registers

... ... section continues....

9. MOV CR opcode table corrected

In Section "MOV—Move to/from Control Registers" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, a correction has been made to the summary table. The table is reproduced below noted by change bars.

MOV—Move to/from Control Registers

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 22 /r	MOV CR0,*r32*	N.E.	Valid	Move *r32* to CR0.
0F 22 /r	MOV CR0,*r64*	Valid	N.E.	Move *r64* to extended CR0.
0F 22 /r	MOV CR2,*r32*	N.E.	Valid	Move *r32* to CR2.
0F 22 /r	MOV CR2,*r64*	Valid	N.E.	Move *r64* to extended CR2.
0F 22 /r	MOV CR3,*r32*	N.E.	Valid	Move *r32* to CR3.
0F 22 /r	MOV CR3,*r64*	Valid	N.E.	Move *r64* to extended CR3.
0F 22 /r	MOV CR4,*r32*	N.E.	Valid	Move *r32* to CR4.
0F 22 /r	MOV CR4,*r64*	Valid	N.E.	Move *r64* to extended CR4.
0F 22 /r	MOV CR8,*r32*	N.E.	N.E.	Move *r32* to CR8.
REX + 0F 22 /r	MOV CR8,*r64*	Valid	N.E.	Move *r64* to extended CR8.
0F 20 /r	MOV *r32*,CR0	N.E.	Valid	Move CR0 to *r32*.
0F 20 /r	MOV *r64*,CR0	Valid	N.E.	Move extended CR0 to *r64*.
0F 20 /r	MOV *r32*,CR2	N.E.	Valid	Move CR2 to *r32*.
0F 20 /r	MOV *r64*,CR2	Valid	N.E.	Move extended CR2 to *r64*.
0F 20 /r	MOV *r32*,CR3	N.E.	Valid	Move CR3 to *r32*.
0F 20 /r	MOV *r64*,CR3	Valid	N.E.	Move extended CR3 to *r64*.
0F 20 /r	MOV *r32*,CR4	N.E.	Valid	Move CR4 to *r32*.
0F 20 /r	MOV *r64*,CR4	Valid	N.E.	Move extended CR4 to *r64*.
0F 20 /r	MOV *r32*,CR8	N.E.	N.E.	Move CR8 to *r32*.
REX + 0F 20 /r	MOV *r64*,CR8	Valid	N.E.	Move extended CR8 to *r64*.[1]

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

... ... section continues...

10. Coverage of PEBS updated

In Section 18.14.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, coverage of PEBS has been updated. Coverage has also been updated in Appendix B of the same volume. See the reproductions of the applicable sections below noted by change bars.

Addition to Chapter 18, Vol. 3B.

18.14.4 Precise Even Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support precise event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor (See Section 18.15.8). The information provides architectural state of the instruction executed immediately after the instruction that caused the event.

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, events that support precise sampling are listed in Table 18-15. The procedure for detecting availability of PEBS is the same as described in Section 18.15.8.1.

Table 18-15. PEBS Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

18.14.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 18-24 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 18-15.

18.14.4.2 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 18.5.2.2, “Debug Store (DS) Mechanism,” for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-16.

Table 18-16. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS	• IA32_MISC_ENABLES.EMON_AVAILABE (bit 7) is set. • IA32_MISC_ENABLES.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (0x38F) with 0. On subsequent entries: • Clear all counters if "Counter Freeze on PMI" is not enabled. • If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[a]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (0x 38E) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (0x 38E) using IA32_CR_PERF_GLOBAL_OVF_CTRL MSR (0x390).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	• Set local enable bit 22 - 1. • Do NOT set local counter PMI/INT bit, bit 20 - 0. • Event programmed must be PEBS capable.	• Set appropriate OVF_PMI bits - 1. • Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (0x38F).	Set each CCCR enable bit 12 - 1.

a. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

Addition to Appendix B, Vol. 3B.

B-1. MSRs in Processors Based on Intel Core Microarchitecture (Contd.)

... table continues...				
3F1H	1009	IA32_PEBS_ ENABLE	Unique	See Section 18.14.4, "Precise Even Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
....table continues...				

11. Missing exception added for MFENCE

In Section "MFENCE—Memory Fence" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, an exception section has been added. The new section is reproduced below.

MFENCE—Memory Fence

... ... more material here ... ...

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.

... ... more material here ... ...

12. IA32_MCG_STATUS information added

In Section 7.8.5 of the *IA-32 Intel® Architecture Software Developer's Manual, Volume 3A*, information has been updated to better reflect the implementation of IA32_MCG_STATUS. This section is reproduced below noted by change bars.

7.8.5 Machine Check Architecture

In the HT Technology context, only the IA32_MCG_STATUS MSR is duplicated for each logical processor. This design is compatible with machine check exception handlers that follow guidelines given in Chapter 14. Note that the MCA specification permits duplication of MSRs other than IA32_MCG_STATUS, but current implementations do not take advantage of this. Software that follows the guidelines in Chapter 14 for machine check exception handlers does not need to be aware of whether an implementation duplicates the other machine check MSRs.

The IA32_MCG_STATUS MSR is duplicated for each logical processor so that its machine check in progress bit field (MCIP) can be used to detect recursion on the part of MCA handlers. In addition, the MSR allows each logical processor to determine that a machine-check exception is in progress independent of the actions of another logical processor in the same physical package.

Because the logical processors within a physical package are tightly coupled with respect to shared hardware resources, both logical processors are notified of machine check errors that occur within a given physical processor. If machine-check exceptions are enabled when a fatal error is reported, all the logical processors within a physical package are dispatched to the machine-check exception handler. If machine-check exceptions are disabled, the logical processors enter the shutdown state and assert the IERR# signal.

When enabling machine-check exceptions, the MCE flag in control register CR4 should be set for each logical processor.

13. Introduction section for CPUID updated

In Section "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, a footnote has been added to clarify behavior in 64-bit processors. The impacted area is reproduced below noted by change bars.

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.[1] The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

.. .. section continues ...

14. Update to CPUID documentation on deterministic cache parameters leaf

In Table 3-12 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, information has been added for CPUID.04H:EAX[Bit 10, Bit 11] values. The impacted part of the table has been reproduced below noted by change bars.

Table 3-12. Information Returned by CPUID Instruction (contd.)

… table continues ….	
	Deterministic Cache Parameters Leaf
04H	**NOTES:** 04H output depends on the initial value in ECX. See also: “INPUT EAX = 4: Returns Deterministic Cache Parameters for each level on page 3-177.
	Deterministic Cache Parameters Leaf
	EAX Bits 4-0: Cache Type Field 0 = Null - No more caches 3 = Unified Cache 1 = Data Cache 4-31 = Reserved 2 = Instruction Cache Bits 7-5: Cache Level (starts at 1) Bits 8: Self Initializing cache level (does not need SW initialization) Bits 9: Fully Associative cache
	Bit 10: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 11: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bits 13-12: Reserved Bits 25-14: Maximum number of threads sharing this cache in a physical package* Bits 31-26: Maximum number of processor cores in the physical package* ** EBX Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*
	ECX Bits 31-00: S = Number of Sets* EDX Reserved = 0 **NOTES:** * Add one to the return value to get the result. **The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
… table continues ….	

15. Updated pseudocode in VMCALL description

In Section “VMCALL—Call to VM Monitor” in Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, the pseudocode has been corrected. See the reproduced segment below noted by change bars.

Operation

IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF (RFLAGS.VM = 1) OR (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF in SMM or if the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail (VMCALL executed in VMX root operation);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 24.16.2
     of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 24.16.6.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*)
    THEN VMfailValid (VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
        THEN
            leave SMM;
            VMfailValid(VMCALL with incorrect MSEG revision identifier);
        ELSE
            read SMM-monitor features field in MSEG (see Section 24.16.6.2,
            in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*);
            IF features field is invalid
                THEN
                    leave SMM;
                    VMfailValid(VMCALL with invalid SMM-monitor features);
                ELSE activate dual-monitor treatment of SMIs and SMM (see Section 24.16.6
                in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*);
            FI;
        FI;
FI;

16. IA32_MC*i*_STATUS figure corrected

In Figure 14.5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, a field definition has been corrected. The figure is reproduced below. See the model-specific error code field.

.

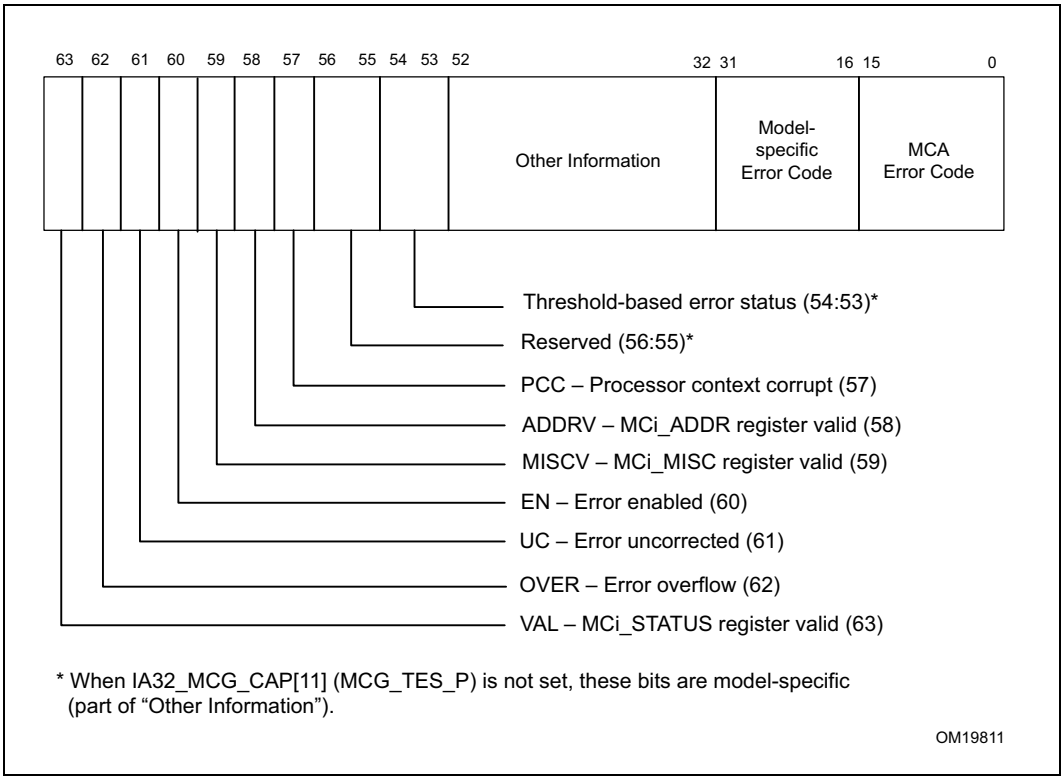

Figure 14-5. IA32_MCi_STATUS Register

17. IA32_MCi_STATUS flag description updated

In Section 14.8.1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the information describing IA32_MCi_STATUS flags has been corrected. This section is reproduced below noted by change bar.

14.8.1 Machine-Check Exception Handler

The machine-check exception (#MC) corresponds to vector 18. To service machine-check exceptions, a trap gate must be added to the IDT. The pointer in the trap gate must point to a machine-check exception handler. Two approaches can be taken to designing the exception handler:

1. The handler can merely log all the machine status and error information, then call a debugger or shut down the system.
2. The handler can analyze the reported error information and, in some cases, attempt to correct the error and restart the processor.

For Pentium 4, Intel Xeon, P6 family, and Pentium processors; virtually all machine-check conditions cannot be corrected (they result in abort-type exceptions). The logging of status and error information is therefore a baseline implementation requirement.

When recovery from a machine-check error may be possible, consider the following when writing a machine-check exception handler:

- To determine the nature of the error, the handler must read each of the error-reporting register banks. The count field in the IA32_MCG_CAP register gives number of register banks. The first register of register bank 0 is at address 400H.
- The VAL (valid) flag in each IA32_MC*i*_STATUS register indicates whether the error information in the register is valid. If this flag is clear, the registers in that bank do not contain valid error information and do not need to be checked.
- To write a portable exception handler, only the MCA error code field in the IA32_MC*i*_STATUS register should be checked. See Section 14.7, “Interpreting the MCA Error Codes,” for information that can be used to write an algorithm to interpret this field.
- The RIPV, PCC, and OVER flags in each IA32_MCi_STATUS register indicate whether recovery from the error is possible. If PCC or OVER are set, recovery is not possible. If RIPV is not set, program execution can not be restarted reliably. When recovery is not possible, the handler typically records the error information and signals an abort to the operating system.
- Correctable errors are corrected automatically by the processor. The UC flag in each IA32_MC*i*_STATUS register indicates whether the processor automatically corrected an error.
- The RIPV flag in the IA32_MCG_STATUS register indicates whether the program can be restarted at the instruction indicated by the instruction pointer (the address of the instruction pushed on the stack when the exception was generated). If this flag is clear, the processor may still be able to be restarted (for debugging purposes) but not without loss of program continuity.
- For unrecoverable errors, the EIPV flag in the IA32_MCG_STATUS register indicates whether the instruction indicated by the instruction pointer pushed on the stack (when the exception was generated) is related to the error. If the flag is clear, the pushed instruction may not be related to the error.

The MCIP flag in the IA32_MCG_STATUS register indicates whether a machine-check exception was generated. Before returning from the machine-check exception handler, software should clear this flag so that it can be used reliably by an error logging utility. The MCIP flag also detects recursion. The machine-check architecture does not support recursion. When the processor detects machine-check recursion, it enters the shutdown state.

18. Instruction summaries fixed for MOVD/MOVQ, PMOVMSKB, PINSRW, PEXTRW

For sections on individual instructions in Chapters 3 and 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A & 2B*, the positioning of REX prefixes in instruction summary tables has been corrected. The applicable tables are reproduced below noted by change bars.

MOVD / MOVQ—Move Doubleword / Move Quadword

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 6E /*r*	MOVD *mm, r/m32*	Valid	Valid	Move doubleword from *r/m32* to *mm*.
REX.W + 0F 6E /*r*	MOVQ *mm, r/m64*	Valid	N.E.	Move quadword from *r/m64* to *mm*.
0F 7E /*r*	MOVD *r/m32, mm*	Valid	Valid	Move doubleword from *mm* to *r/m32*.
REX.W + 0F 7E /*r*	MOVQ *r/m64, mm*	Valid	N.E.	Move quadword from *mm* to *r/m64*.
66 0F 6E /*r*	MOVD *xmm*, *r/m32*	Valid	Valid	Move doubleword from *r/m32* to *xmm*.
REX.W 66 0F 6E /*r*	MOVQ *xmm*, *r/m64*	Valid	N.E.	Move quadword from *r/m64* to *xmm*.
66 0F 7E /*r*	MOVD *r/m32*, *xmm*	Valid	Valid	Move doubleword from *xmm* register to *r/m32*.
REX.W 66 0F 7E /*r*	MOVQ *r/m64*, *xmm*	Valid	N.E.	Move quadword from *xmm* register to *r/m64*.

Text omitted here.....

PMOVMSKB—Move Byte Mask

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F D7 /*r*	PMOVMSKB *r32*, *mm*	Valid	Valid	Move a byte mask of *mm* to *r32*.
REX.W + 0F D7 /*r*	PMOVMSKB *r64*, *mm*	Valid	N.E.	Move a byte mask of mm to the lower 32-bits of r64 and zero-fill the upper 32-bits.
66 0F D7 /*r*	PMOVMSKB *r32*, *xmm*	Valid	Valid	Move a byte mask of *xmm* to *r32*.
66 REX.W 0F D7 /*r*	PMOVMSKB *r64*, *xmm*	Valid	N.E.	Move a byte mask of xmm to the lower 32-bits of r64 and zero-fill the upper 32-bits.

Text omitted here.....

PINSRW—Insert Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F C4 /*r* ib	PINSRW *mm*, *r32/m16*, *imm8*	Valid	Valid	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8*
REX.W + 0F C4 /*r* ib	PINSRW *mm*, *r64/m16*, *imm8*	Valid	N.E.	Insert the low word from *r64* or from *m16* into *mm* at the word position specified by *imm8*
66 0F C4 /*r* ib	PINSRW xmm, *r32/m16*, imm8	Valid	Valid	Move the low word of *r32* or from *m16* into xmm at the word position specified by *imm8*.
66 REX.W 0F C4 /*r* ib	PINSRW xmm, *r64/m16*, imm8	Valid	N.E.	Move the low word of *r64* or from *m16* into xmm at the word position specified by *imm8*.

Text omitted here.....

PEXTRW—Extract Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F C5 /*r* ib	PEXTRW *r32*, *mm*, *imm8*	Valid	Valid	Extract the word specified by *imm8* from *mm* and move it to *r32*, bits 15-0. Zero-extend the result.
REX.W + 0F C5 /*r* ib	PEXTRW *r64*, *mm*, *imm8*	Valid	N.E.	Extract the word specified by *imm8* from *mm* and move it to *r64*, bits 15-0. Zero-extend the result.
66 0F C5 /*r* ib	PEXTRW *r32*, *xmm*, *imm8*	Valid	Valid	Extract the word specified by *imm8* from *xmm* and move it to *r32*, bits 15-0. Zero-extend the result.
66 REX.W 0F C5 /*r* ib	PEXTRW *r64*, *xmm*, *imm8*	Valid	N.E.	Extract the word specified by *imm8* from *xmm* and move it to *r64*, bits 15-0. Zero-extend the result.

Text omitted here.....

19. Correction to microcode update documentation

In Section 9.11.6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, additions made pertaining to 64-bit support. The section is reproduced below noted by change bars.

9.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a Pentium 4, Intel Xeon, or P6 family processor. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

Example 9-8. Assembly Code Example of Simple Microcode Update Loader

```
mov   ecx,79h              ; MSR to read in ECX
xor   eax,eax              ; clear EAX
xor   ebx,ebx              ; clear EBX
mov   ax,cs                ; Segment of microcode update
shl   eax,4
mov   bx,offset Update     ; Offset of microcode update
add   eax,ebx              ; Linear Address of Update in EAX
add   eax,48d              ; Offset of the Update Data within the Update
xor   edx,edx              ; Zero in EDX
WRMSR                      ; microcode update trigger
```

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, pages that are currently present must map the update data.
- The microcode update data requires a 16-byte boundary alignment.

Section continues, omitted material starts here.....

20. PSHUFB compiler intrinsic fixed

In Section "PSHUFB — Packed Shuffle Bytes" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, a compiler intrinsic has been corrected. The new subsection is reproduced below noted by change bar.

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)

PSHUFB __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)